SCO UNIX® System V/386

Development System

C Language Guide

The Santa Cruz Operation, Inc.

SCO Document Number: 6-26-89-6.0/3.2.0

SCO UNIX® System V/386

Development System

C User's Guide

The Santa Cruz Operation, Inc.

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

Chapter 1

Introduction

Overview

The C language is a powerful general-purpose programming language that can generate efficient, compact, and portable code. The Microsoft® C Optimizing Compiler (**cc**) for the UNIX System V® operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*. Microsoft is actively involved in the development of the ANSI (American National Standards Institute) standard for the C language; this version of Microsoft C for UNIX System V anticipates and conforms to the forthcoming standard in many areas.

The Microsoft C Compiler offers several important features to help you increase the efficiency of your C programs. You can choose among five standard memory models (small, medium, compact, large, and huge) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the C Compiler allows you to "mix" memory models by using special declarations in your program.

The C language itself does not provide such standard features as input and output capabilities and string-manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the C Compiler.

Compared with other programming languages, Microsoft C is extremely flexible concerning data conversions and nonstandard constructions. The C Compiler offers several levels of warnings to help you control this flexibility; programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. An experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions. For more information about this feature, see the "Compiling with the cc Command" chapter in this guide.

1 About This Guide

This guide explains how to use the C Compiler to compile, link, and run C programs on UNIX System V. The guide assumes that you are familiar with the C language and with UNIX System V, and that you know how to create and edit a C-language source file on your system.

If you have questions about the C language, turn to the *C Language Reference* included in this package. The *C Library Guide* documents the run-time library routines you can use in your C programs.

The remaining chapters of the *C User's Guide* are described below:

Chapter 2, "Compiling with the cc Command," describes how to compile a program using the **cc** compiler driver. This chapter describes the options most commonly used to control preprocessing, compiling, and output of files.

Chapter 3, "Linking with the cc Command," describes how to link object files using the **cc** command. This chapter explains how the linker searches for libraries, shows how to specify libraries for linking, and describes the linker options that can be used for C programs.

Chapter 4, "Running C Programs on UNIX System V," explains how to run your executable program file and discusses features specific to the UNIX System V implementation of C. This chapter tells how to pass data from UNIX System V to a program at execution time and how to return an exit code from your program to UNIX System V.

Chapter 5, "Working with Memory Models," describes methods of managing memory models. These methods are useful for writing programs that use more than 64K (kilobytes) of code or data. This chapter also discusses "mixed-model" programming (combining features from the five standard memory models).

Chapter 6, "Improving Program Speed," gives suggestions and hints for maximizing program speed.

Chapter 7, "Object and Executable File Formats," describes the system architecture of the 80x86 microprocessor family, the object module format that the C compiler follows, and the format of the **x.out** file in a segmented environment.

Chapter 8, "C Language Compatibility with Assembly Language," describes how you can embed assembly-language subroutines within C-language programs.

Chapter 9, "Error Processing," describes how to process errors detected in calls to the C library routines and explains the functions and variables a program may use to respond to these errors.

Chapter 10, "The Common Object File Format (COFF)," describes the features and contents of *COFF* files.

Appendix A, "Converting from Previous Versions of the Compiler," summarizes the differences between Version 5.1 of the C Compiler and previous versions. This appendix gives instructions for converting programs written for versions prior to 5.1 to the format accepted by Version 5.1.

Appendix B, "Writing Portable Programs," lists some of the C-language features that are implementation-dependent, and offers suggestions for increasing program portability.

Appendix C, "Writing Programs for Read-Only Memory," gives information about modifying start-up code and initializing floating-point support for programs that will be put in read-only memory.

Appendix D, "Error Messages and Exit Codes," lists and describes the error messages and exit codes generated by the C Compiler and by the **cc** command. It also lists and explains run-time error messages produced by executable programs written in C.

1

New Features

Several useful features have been added to Version 5.1 of the C Compiler. This section summarizes features added since Version 5.0. For information about differences between Version 5.1 and versions prior to 5.0, see the ''Converting from Previous Versions of the Compiler'' appendix in this guide.

New features include the following:

Feature	Description

New **cc** options

Option	Action
-S	Generates an assembly-language source file for the Macro Assembler, *masm*(CP).
-xenix	Produces object and/or executable files using the Intel Object Module Format (OMF).
-x2.3	Produces object and/or executable files using the Intel Object Module Format (OMF) and the XENIX System V/Release 2.3 run-time library.

New pragmas

Pragma	Action
comment	Places a comment record in the object file.
Data_seg	Specifies the data-segment name used by functions that load their own data segments. The named segment also contains all data that would normally be allocated in the **DATA** segment.
linesize	Sets the number of characters per line in the source listing.

message	Sends a message to the standard output without terminating the compilation.
page	Places a formfeed character(s) in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing. Places a comment record in the object file.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

1 Notational Conventions

The following notational conventions are used throughout this guide:

Example of Convention	Description of Convention
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on the screen or by a printer. For example, the following command line is printed in this special typeface: `cc -Foout.o -DTRUE=1 file.c` When this command line is discussed in text, items appearing on the command line, such as *out.o*, also appear in the special typeface.
Language elements	Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives. Examples are shown below: **+= #if defined( ) int** **if -Fa fopen** **main sizeof**
ENVIRONMENT, VARIABLES, and **MACROS**	Bold capital letters are used for environment variables, symbolic constants, and macros.
placeholders	Words in italics are placeholders, representing a variable that you must supply in command-line examples, option specifications, and in the text. Consider the following option: **-H** *number*

Note that *number* is italicized to indicate that it represents a general form for the **-H** option. In an actual command, you would supply a particular number for the placeholder *number*.

Occasionally, italics are also used to emphasize particular words in the text.

Missing code

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;
.
.
.
*pc++;
```

[*optional items*]

Brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

-D*identifier*[=[*string*]]

The placeholder *identifier* indicates that you must supply an identifier when you use the **-D** option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used in C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

Repeating elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, in the Bourne shell, several paths can be specified in the **PATH** command, as shown in the following syntax:

PATH[=]*path*[;*path*]...

{*choice1*|*choice2*}

Braces and a vertical bar indicate that you have a choice of two or more items. Braces enclose the choices, and vertical bars separate them. You must choose one of these items unless all of them are also enclosed in square brackets.

For example, the **-W** (warning-level) compiler option has the following syntax:

-W {0 | 1 | 2 | 3}

You can use **-W1**, **-W2**, or **-W3** to display different levels of warning messages or **-W0** to suppress all warning messages.

''Defined terms''

Quotation marks set off terms defined in the text. For example, the term ''far'' appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than `` ''. For example, a C string used in an example would be shown in the following form:

```
"abc"
```

KEY+KEY

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (CTRL+C).

Books about C

The manuals in this documentation package provide a complete programmer's reference for C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one or more of the following books:

Hancock, Les, and Morris Krieger. *The C Primer*. New York: McGraw-Hill Book Co., Inc., 1982.

Hansen, Augie. *Proficient C*. Bellevue, Washington: Microsoft Press, 1986.

Harbison, Samuel P., and Greg L. Steele. *C: A Reference Manual*. Englewood Cliffs, New Jersey: Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas. *Learning to Program in C*. Cardiff, New Jersey: Plum Hall, Inc., 1983.

Schildt, Herbert. *C Made Easy*. Berkeley, California: Osborne McGraw Hill, 1985.

Schustack, Steve. *Variations in C*. Bellevue, Washington: Microsoft Press, 1985.

These books are listed for your convenience only.

Chapter 2

Compiling with the cc Command

Introduction

This chapter explains how to compile and link using the **cc** command and discusses commonly used **cc** options. The **cc** command is the only command you need to compile and link your C source files. The **cc** command executes the three compiler passes, then automatically invokes the linker, **ld**, to link your files.

Using the **cc** options described in this chapter, you can control and modify the tasks performed by the command. For example, you can direct **cc** to create an object-listing file or a preprocessed listing. Options also let you give information that applies to the compilation process; you can specify the definitions for manifest (symbolic) constants and macros, and the kinds of warning messages you want to see.

The **cc** command automatically optimizes your program. You never have to give an optimizing instruction unless you want to change the way **cc** optimizes, request more sophisticated optimizations, or disable optimization altogether. For more information on these choices, see the "Optimizing" section in this chapter.

"The Basics: Compiling and Linking C Programs" explains the basic use of the **cc** command to produce an executable program.

"Using cc Options," describes the **cc** options.

For information about linking object files and libraries using the **cc** command, see the "Linking with the cc Command" chapter of this guide.

For a discussion of the **cc** options that control memory models, see the "Working with Memory Models" chapter in this guide.

For a summary of the **cc** command and its options, see the *C Language Reference*.

The Basics: Compiling and Linking C Programs

This section explains how to use **cc** to compile and link C programs and discusses the rules and conventions that apply to file names and options used with **cc**.

The cc Command

The **cc** command has the following form:

cc [*option*]... *file*... [*option*... *file*...]...[-link[*link-libinfo*]]

Each *option* is one of the command-line options described in the ''Using cc Options'' section, the ''Working with Memory Models'' chapter, or the ''Improving Program Speed'' chapter of this guide.

Each *file* names a source or object file to be processed or a library to be searched at link time. See the description on ''Specifying Source and Object Files'' later in this section for information about specifying source and object files.

The **cc** command automatically specifies the appropriate library to be used during linking. You can use the **-link** option with the optional *link-libinfo* field to specify additional or different libraries, library search paths, and options to be used during linking. You can also specify linker options in the *linkoptions* field. For information about specifying different libraries and linker options, see the ''Linking with the cc Command'' chapter of this guide.

You can give any number of options, file names, and library names on the command line, provided that the command line does not exceed 128 characters.

COFF and OMF

This version of the C Compiler can produce object and/or executable files that use either of two different binary file formats: COFF (Common Object File Format) and OMF (Intel Object Module Format). COFF is the most widely used binary file format. OMF files are produced using the

-xenix option with the compiler. SCO UNIX System V can execute either file format by reading the file header and acting accordingly. Certain system calls behave differently in OMF files because they follow UNIX System V rather than XENIX system conventions. The COFF and OMF formats are described by their corresponding header files: **/usr/include/a.out.h** and **/usr/include/sys/x.out.h** respectively.

Note

The default file name produced by the linker is *a.out* regardless of the actual file format used. Any mention of *x.out* in this guide is referring only to the *format* of OMF executable files.

Table 2.1 shows the tools used with various options to the Microsoft C Compiler, and the type of object/executable file that results.

Table 2.1

Options, Tools, and Resulting Files

Option	MS C Compiler	Assembler	Link Editor	obj format
none	MS	masm	*ld*	COFF
-c	Creates linkable x.out object file	masm	n/a	COFF
-S	Creates assembly source listing	masm	n/a	COFF
-Fa	Creates assembly source listing	masm	*ld*	COFF
-xenix	Creates XENIX programs in OMF format	masm	*ld*	OMF
-S -xenix	Creates XENIX assembly source listing	masm	n/a	OMF
-Fa -xenix	Creates XENIX assembly source listing	masm	*ld*	OMF

Specifying Source and Object Files

The **cc** command can process source files, object files, library files, or any combination of these. It uses the file-name extension (the period plus any letters that follow it) to determine what kind of processing the file needs, as shown in the following list:

- If the file has a **.c** extension, **cc** compiles the file.

- If the file has a **.o** extension, **cc** processes the file by invoking the linker.
- If the file has a **.a** extension, **cc** assumes the file is a library and passes it to the linker to be searched, unless the **-c** option is given to suppress linking. For a description of the **-c** option, see the section on ''Compiling without Linking'' under the section ''Using cc Options.''
- If the file has the **.asm** extension, it is passed to **masm**.
- If the extension is omitted,
  cc assumes an extension of **.o**. If the extension is anything other than **.c** , **.o** , or **.a** , **cc** assumes the file is an object file unless the file name is specified in association with the **-Tc** option. If the file name is specified with the **-Tc** option, **cc** assumes the file is a C source file. For a description of the **-Tc** option, see the section on ''Specifying Source Files'' under the section ''Using cc Options.''

Examples

```
cc a.c b.c c.o d.o
```

This command line compiles the files *a.c* and *b.c*, creating object files named *a.o* and *b.o*. These object files are then linked with the object files *c.o* and *d.o* to form an executable file named *a.out*.

```
cc a.c b.c c.o -Tcd.src
```

This command performs the same operations as the preceding command line, except that the **-Tc** option indicates that *d.src* is a source file, not an object file. Thus, the files *a.c*, *b.c*, and *d.src* are compiled, creating object files named *a.o*, *b.o*, and *d.o*. These object files are then linked with *c.o* to form an executable file named *a.out*.

Creating Executable Files

When **cc** compiles source files, it creates object files. By default, these object files use the COFF format and have the same base names as the corresponding source files, but with the extension **.o** instead of **.c**. (The base name of a file extension is the portion of the name preceding the period, but excluding the path specification, if any.) After compilation, **cc** runs a conversion program, **cvtomf**, over the object file to convert it into COFF format. For more information about the **cvtomf** conversion program, refer to the manual page **cvtomf (C)**. The converted object file can

now be linked using the AT&T link editor, **ld**. The **-xenix** option suppresses the conversion.

Unless the **-c** option is given, **cc** links these object files, along with any **.o** files you give on the command line, to form an executable file. If only **.o** files are given on the command line, **cc** skips the compilation stage and simply links the files.

Using cc Options

The **cc** command offers a large number of command options to control and modify the compiler's operation. Options begin with a dash (-) and contain one or more letters.

Options can appear anywhere on the **cc** command line. In general, an option applies to all files that follow it on the command line, and it does not affect files preceding it. However, not all options follow this rule; see the discussion of a particular option for information on its behavior. Keep in mind that **cc** options apply only to the compilation process. Unless specifically noted, options do not affect any object files given on the command line. The remainder of this section describes many of the options applicable to **cc**. For a concise list of all possible options, refer to the manual page, **cc(CP)**.

Setting Processor and Memory Model (-M)

The **-M** option sets the program configuration. This configuration defines the program's memory model, word order, and data threshold. It also enables C-language enhancements such as the use of the full 286 instruction set and special keywords.

```
cc -Mstring special.c
```

The *string* contains the argument that defines the configuration. It may be any combination of the following (though **s, m, c, l, h,** and **0, 1, 2, 3** are mutually exclusive):

s	Create a small model program. This is the default.
m	Create a middle model program.
c	Create a compact model program.
l	Create a large model program.
h	Create a huge model program.

e Enable the keywords: **far, near, huge, pascal** and **fortran**. Also enables certain non-ANSI extensions necessary to ensure compatibility with existing versions of the C compiler.

0 Use only 8086 instructions for code generation. This is the default on 8086/80186/80286 systems.

1 Use the extended 80186 instruction set.

2 Use the extended 80286 instruction set.

3 Use the extended 80386 instruction set. This is the default on 80386 systems.

t*num* Causes all static and global data items whose size is greater than *num* bytes to be allocated to a new data segment. *Num*, the data "threshold," defaults to 32,767. This option can only be used in large model programs (**-Ml**). Its main use is to move data out of the near data segment to allow room for the stack.

```
cc -Ml -Mt12 recursive.c
```

d Do not assume (during compilation) that the registers **SS** and **DS** will have the same contents at run time.
Warning: This option has no library or run-time support on UNIX System V. It will **not** cause the stack to be put in a separate segment. It may be of use for DOS cross-development.

-M3 is the default on 80386 systems. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80186/80188 or 80286 instruction set.

Note

The **m**, **c**, **l**, **h**, **b**, **t**, and **d** arguments are not compatible with the **-M3** option. The **s** and **e** arguments are compatible with **-M0**, **-M1**, **-M2**, or **-M3**.

For a complete description of memory models and segment options, see the ''Working with Memory Models'' chapter in this guide.

The memory-model option you choose determines the name of the standard libraries that **cc** places in the object file it creates. These libraries are then considered the default libraries, since the linker searches for them by default.

Table 2.2 shows each memory-model option and the corresponding library name that **cc** embeds in the object file.

Table 2.2

cc Options and Default Libraries

Memory-Model Option	Default Libraries
-Ms	**Slibc.a** **Slibcfp.a**
-Mm	**Mlibc.a** **Mlibcfp.a**
-Mc	**Clibc.a** **Clibcfp.a**
-Ml or **-Mh**	**Llibc.a** **Llibfp.a**

Specifying Source Files (-Tc)

Option

-Tc *sourcefile*

The **-Tc** option tells the **cc** command that the given file is a C source file. One or more spaces can appear between **-Tc** and the source-file name.

If this option does not appear, **cc** assumes that files with the extension **.c** are C source files, files with the extension **.a** are libraries, and files with any other extension or with no extension are object files. If you use the **-Tc** option, **cc** treats the given file as a C source file, regardless of its extension. A separate **-Tc** option must appear for each source file that has an extension other than **.c** .

If you have to specify more than one source file with an extension other than **.c** , you must specify each source file in a separate **-Tc** option.

Example

```
cc main.c -Tc test.prg -Tc collate.prg print.prg
```

In this example, the **cc** command compiles the three source files *main.c*, *test.prg*, and *collate.prg*. Since the file *print.prg* is given without a **-Tc** option, **cc** treats it as an object file. Thus, after compiling the three source files, **cc** links the object files *main.o*, *test.o*, *collate.o*, and *print.prg*.

Compiling without Linking (-c)

Option

-c

The **-c** (for ''compile-only'') option suppresses linking. Source files given on the command line are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified on the command line are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The **-c** option applies to the entire **cc** command line, regardless of the option's position in the command line.

Example

```
cc -c *.c
```

This command line compiles, but does not link, all files with the extension **.c** in the current working directory.

Naming the Object File (-Fo)

Option

-Fo*objfile*

By default, **cc** gives each object file it creates the base name of the corresponding source file plus the extension **.o**. The **-Fo** option lets you give different names to object files or create them in a different directory. If you are compiling more than one source file, you can give the **-Fo** option for each source file to rename the corresponding object file.

Keep the following rules in mind when using this option:

- The *objfile* argument must appear immediately after the option, with no intervening spaces.
- Each **-Fo** option applies to the next source file that appears on the command line after the option.

You are free to supply any name and any extension you like for the *objfile*. However, it is recommended that you use the conventional **.o** extension because the linker uses **.o** as the default extension when processing object files.

If you use the **-Fo** option (that is, if you do not give an object file name with a base and an optional extension), **cc** names the object files according to the following rule:

- If you give a directory specification following the **-Fo** option, **cc** creates the object file in the given directory and uses the default file name (the base name of the source file plus **.o**). Otherwise, *objfile* is created in the current directory. A **.o** extension is added if no extension is given.

To give a directory specification, it must end with a forward slash (/) so that **cc** can distinguish between a directory specification and a file name.

Example

```
cc -Fo/object1/ this.c that.c -Fo/src/newthose those.c
```

In this example, the first **-Fo** option tells the compiler to create, in the */object1* directory, the object file *this.o* (created as a result of compiling *this.c*). The compiler also creates, in the current directory, the object file *that.o* (created as a result of compiling *that.c*). The second **-Fo** option tells the compiler to create the object file named *newthose.o* (created as a result of compiling *those.c*) in the */src* directory.

Naming the Executable File (-Fe) (-o)

Option

-Fe*exefile*
-o *exefile*

By default, **cc** gives the name *a.out* to the executable file. In UNIX System V, **-Fe** and **-o** are the same, except syntactically. The file name must come immediately after **-Fe**, whereas blanks are permitted between **-o** and the file name. Either option lets you give the executable file a different name or create it in a different directory.

Since **cc** creates only one executable file, you can give the **-Fe** option anywhere on the command line. If more than one **-Fe** option appears, **cc** gives the executable file the name specified in the last **-Fe** option on the command line.

The **-Fe** option applies only in the linking stage. If you specify the **-c** option to suppress linking, **-Fe** has no effect.

Examples

```
cc -Fe/bin/process *.c
cc -o /bin/process *.c
```

These examples compile and link all source files with the extension **.c** in the current working directory. The resulting executable file is named *process.out* and is created in the directory */bin*.

Creating Listings

A number of options are available with the **cc** command for creating listings. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set the title and subtitle of the source listing from the command line and control the length of source-listing lines and pages.

These options are described in the following sections.

Note

Listings produced by the **cc** command may contain names that begin with more than one underscore (for example, *__chkstk*) or that end with the suffix *QQ*. Names that use these conventions are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *C Library Guide*. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with names reserved by the compiler.

Types of Listings (-Fs, -Fl, -Fa, -Fc, -Fm -S)

Options

-Fs[*listfiles*]	Source listing
-Fl[*listfile*]	Object listing
-Fa[*listfile*]	Assembly listing
-Fc[*listfile*]	Combined source and object listing
-Fm[*mapfile*]	Map file that lists segments, in order
-S	Assembly listing

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see the section on "Formats for Listings."

When using an option described in this section, the *listfile* argument, if given, must follow the option immediately, with no intervening spaces. The *listfile* may be a file specification or a path specification. It may also be omitted.

Note

When you give just a path specification as the *listfile* argument, the path specification must end with a forward slash (/) so that **cc** can distinguish it from an ordinary file name.

When you give a path specification as the argument to a listing option, or if you omit the argument altogether, **cc** uses the default file name for the listing type. Table 2.3 gives the default names used for each type of listing. The table also shows the default extensions, which are used when you give a file-name argument that lacks an extension.

Table 2.3
Default File Names and Extensions

Option	Listing Type	Default File Name[1]	Default Extension[2]
-Fs	Source	Base name of source file plus **.S**	**.S**
-Fl	Object	Base name of source file plus **.L**	**.L**
-Fa	Assembly (masm)	Base name of source file plus **.asm**	**.asm**
-Fc	Combined source-object	Base name of source file plus **.L**	**.L**
-Fm	Map	Prints to standard output.	
-S	Assembly (masm)	Base name of source file plus **.asm**	**.asm**

Notes:

1 The default file name is used when the option is given with no argument or with a path specification as the argument.

2 The default extension is used when a file name lacking an extension is given.

3 The assembly-language listing produced by the **-Fa**, **-Fc**, and **-S** options uses **masm** directives.

4 The **-Fa** and **-S** options produce the same output, except that you cannot specify the list file with the -S option.

Since you can process more than one file at a time with the **cc** command, the order in which you give listing options and the kind of argument you give for each option (file specification or path specification) affect the result. Table 2.4 summarizes the effects of each option with each type of argument.

Table 2.4
Arguments to Listing Options

Option	File-Name Argument	Path Argument[1]	No Argument
-Fa, -Fc, -Fl, -Fs	Creates a listing for the next source file on the command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
-Fm	Uses given file name for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name
-S	File name argument is not allowed	Path argument is not allowed	Uses default name

Notes:

1 When you give just a path specification as the argument, the path specification must end with a forward slash (/) so that cc can distinguish it from an ordinary file name.

Only one type of object or assembly listing can be produced for each source file. The **-Fc** option overrides the **-Fa** and **-Fl** options and produces a combined listing. If you apply both the **-Fa** and the **-Fl** options to one source file, only the last listing specified on the command line is produced. If you specify both the **-Fa** and the **-Fs** options to one source file, a combined listing is produced. The **-Fs** option may be used with any other option.

Note

The **cc** command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the **-Fc** option to mingle the source and assembly codes. To produce a listing without optimizing, use the **-Od** option (discussed in ''Preparing for Debugging'' later in this section) with the listing option.

The map file is produced during the linking stage. If linking is suppressed with the **-c** option, the **-Fm** option has no effect.

Examples

```
cc -Fshello.src -Fchello.cmb hello.c
```

In this example, **cc** creates a source listing called *hello.src* and a combined source and object listing called *hello.cmb*. The object file has the default name *hello.o*. However, it is removed if the link was successful.

```
cc -Fshello.src -Fshello.lst -Fchello.cod hello.c
```

This command produces a source listing called *hello.lst* rather than *hello.src*, since the last name provided has precedence. This example also produces a combined source and object listing file named *hello.cod*. The object file in both of these examples has the default name *hello.o*.

Setting Titles (-St) and Subtitles (-Ss)

Options

-St "*title*"
-Ss "*subtitle*"

The **-St** and **-Ss** options set the title and subtitle, respectively, for source listings. The quotation marks (" ") around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between **-St** or **-Ss** and its argument is optional.

The title appears in the upper left corner of each page of the source listing. The subtitle appears below the title.

The **-St** or **-Ss** option applies to the remainder of the command line or until the next occurrence of **-St** or **-Ss** on the command line. These options do not cause source listings to be created. They take effect only when the **-Fs** option is also used to create a source listing.

Examples


```
cc -St "Income Tax" -Ss 4-14 -Fs tax*.c
```

This command compiles and links all source files beginning with *tax* and ending with the default extension (*.c*) in the current working directory. Each page of the source listing contains the title *Income Tax* in the upper left corner. The subtitle *4-14* appears below the title on each page.

```
cc -c -Fs -Fa -St"Calc Prog" -Ss"count" ct.c -Ss"sort" srt.c
```

In this command, **cc** compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title *Calc Prog*.

Formats for Listings

The rest of this section describes and shows examples of the five types of listings available with the **cc** command. For information on how to create these listings, see ''Types of Listings'' earlier in this section.

Source Listing

Source listings are helpful for debugging programs as they are being developed. These listings are also useful for documenting the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with any diagnostic messages that were generated. If the source file compiles with no errors more serious than warning errors, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each function. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of the segment sizes in your program. This summary is useful for analyzing the program's memory requirements.

Any error messages that occurred during compilation appear in the listing after the line that caused the error, as shown in the following example:

```
 1  char hexvalue[10];
 2
 3  main()
 4  {
 5    long htoi();
 6    printf("Please enter the hex value you want to convert:\n");
 7    scanf("%s", hexvalue);
 8    printf("The integer value of the hex value is %ld\n", htoi(hexvalue));
 9  }
10
11   long htoi(hexvalue)
12   char *hexvalue;
13   {
14       register char *ptr=hexvalue;
15       int i=0;
16       long n=0;
17       long exp16();
18       while (*ptr != '\0') {
19        if (*ptr >= 'a' && *ptr <= 'f')
20                *ptr -= 87;
21        else if (*ptr >= 'A' && *ptr <= 'F')
22                  *ptr -= 55;
23               else
24                       *ptr -= 48;
25        ptr+;
 bomb.c(25) : error C2059: syntax error : ';'
26       }
```

The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing.

The following example shows the source listing for a simple C program:

```
Hex to ASCII                                                    PAGE   1
2/25/87                                                         02-25-87
                                                                10:44:23
 Line#  Source Line                  C Optimizing Compiler Version 5.10
  1  char hexvalue[10];
  2
  3  main()
  4  {
  5    long htoi();
  6    printf("Please enter the hex value you want to convert:0);
  7    scanf("%s", hexvalue);
  8    printf("The integer value of the hex value is %ld0, htoi(hexvalue));
  9  }
 10
 11  long htoi(hexvalue)
 12  char *hexvalue;
 13  {
 14      register char *ptr=hexvalue;
 15      int i=0;
 16      long n=0;
 17      long exp16();
 18      while (*ptr != ' ') {
 19       if (*ptr >= 'a' && *ptr <= 'f')
 20                  *ptr -= 87;
 21       else if (*ptr >= 'A' && *ptr <= 'F')
 22                   *ptr -= 55;
 24                     *ptr -= 48;
 25       ptr++;
 26      }
 27      ptr -= 1;
 28      while (ptr>=hexvalue)
 29         {
 30    n+= (*ptr*exp16(i));
 31          i++;
 32          ptr--; 33          }
 34      return(n);
 35  }
 htoi  Local Symbols
Name    Class   Type      Size   Offset  Register
i . . . . . auto                          -0008
ptr . . . . . auto                        ***       si
n . . . . . auto                          -0004
hexvalue. . . . . param                    0004
 36
 37  long exp16(exp)
 38  int exp;
 39  {
 40          long result=1;
 41   int j;
 42   for (j=1; j<=exp; j++)
 43         result *= 16;
 44   return(result);
 45 }
```

```
Hex to A
2/25/87                                                    02-25-87
                                                           10:44:23

                                C Optimizing Compiler Version 5.10

exp16  Local Symbols

Name                    Class   Type        Size    Offset  Register

j . . . . . . . . . . . auto                        -0006
result. . . . . . . . . auto                        -0004
exp . . . . . . . . . . param                        0004


Global Symbols

Name                  Class   Type                Size    Offset

exp16 . . . . . . . . global  near function       ***      00ae
hexvalue. . . . . . . common  struct/array         10       ***
htoi. . . . . . . . . global  near function       ***      0038
main. . . . . . . . . global  near function       ***      0000
printf. . . . . . . . extern  near function       ***       ***
scanf . . . . . . . . extern  near function       ***       ***

Code size = 00e8 (232)
Data size = 005f (95)
Bss size  = 0000 (0)

No errors detected
```

At the end of each function, a table of local symbols is given, as shown in the following example for the function *htoi*:

```
htoi  Local Symbols

Name                    Class   Type        Size    Offset  Register

i . . . . . . . . . . . auto                        -0008
ptr . . . . . . . . . . auto                        ***       si
n . . . . . . . . . . . auto                        -0004
hexvalue. . . . . . . . param                        0004
```

The following list shows the contents of each column in the symbol table:

Column	Contents
Name	The name of each local symbol in the function.
Class	Either *auto* if the symbol is a nonstatic local variable, or *param* if the symbol is a formal parameter.
Offset	The symbol's offset address relative to the frame pointer (that is, the **BP** register). The *Offset* number is positive for *param* symbols and negative for *auto* symbols with **auto** storage class.
Register	Blank unless the variable is stored in a register, in which case, this column indicates the register (**SI** or **DI**).

At the end of the source code, a table of global symbols is given, as shown in the following example:

```
Name                 Class   Type              Size   Offset

exp16 . . . . . . . global  near function      ***     00ae
hexvalue. . . . . . common  struct/array        10      ***
htoi. . . . . . . . global  near function      ***     0038
main. . . . . . . . global  near function      ***     0000
printf. . . . . . . extern  near function      ***      ***
scanf . . . . . . . extern  near function      ***      ***
```

The following list shows the contents of each column:

Column	Contents
Name	Each global symbol, external symbol, and statically allocated variable declared in the source file.
Class	Either *global*, *common*, *extern*, or *static*, depending on how the symbol was defined in the source file.
Type	A simplified version of the symbol's type as declared in the source file.

For functions, this entry is either *near function* or *far function*, depending on which memory model was used and how the function was declared. For a pointer, this entry is *near pointer*, *far pointer*, or *huge pointer*. For enumeration variables, this entry is *int*. For structures, unions, and arrays, this entry is *struct/array*.

Size Used only for variables. Specifies the number of bytes of storage allocated for the variable. Since the amount of storage allocated for an external array may not be known, its *Size* entry may be undefined.

Offset Used only for symbols with an entry of *global* or *static* in the *Class* column.

For variables, this entry gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker usually combines several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables. For functions, this entry gives the relative offset of the start of the function in the logical code segment. For small-model programs, the linker combines logical code into a single physical segment, so this entry is useful for determining the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, this entry gives the actual offset of the function in its run-time code segment.

The last table in the source listing shows the segments used and their size, as in the following example:

```
Code size = 0103 (259)
Data size = 005f (95)
Bss size  = 0000 (0)
```

The number of bytes in each segment is given first in hexadecimal, and then in decimal (in parentheses).

Object Listing

The **-Fl** option produces an object listing. The object listing contains the instruction encoding and assembly code for your program. The line numbers are shown in the listing as comments. The instruction encoding is on the left and the assembly code on the right, as shown in the following example:

```
; Line 4
      PUBLIC _main
_main PROC NEAR
      *** 000000   55                push bp
      *** 000001   8b ec         mov  bp,sp
      *** 000003   33 c0         xor  ax,ax
      *** 000005   e8 00 00           call __chkstk
; Line 6
      *** 000008   b8 00 00           mov  ax,OFFSET DGROUP:$S G12
      *** 00000b   50                 push ax
      *** 00000c   e8 00 00           call _printf
      *** 00000f   83 c4 02           add  sp,2
```


Assembly Listing

The **-Fa** and **-S** options produce an assembly listing using directives suitable for assembly using the Macro Assembler, **masm**. It contains the assembly code corresponding to your C source file, as shown in the following example:

```
; Line 4
      PUBLIC _main
_main PROC NEAR
      push  bp
      mov   bp,sp
      xor   ax,ax
      call  __chkstk
; Line 6
      mov   ax,OFFSET DGROUP:$SG12
      push  ax
      call  _printf
      add   sp,2
```

Note that the example shows the same code as in the object listing example, except that the instruction encoding is omitted.

The listing generated by the **-Fa** option in Versions 5.0 and later of the C Compiler can be used as input to **masm**.

Combined Source and Object Listing

The **-Fc** option produces a combined source and object listing. This shows each line of your source program followed by the corresponding line (or lines) of machine instructions, as shown in the following example:

```
_TEXT      SEGMENT
;|*** char hexvalue[10];
;|***
;|*** main()
;|*** {
; Line 4
      PUBLIC _main
_main PROC NEAR
      *** 000000   55                push bp
      *** 000001   8b ec        mov  bp,sp
      *** 000003   33 c0        xor  ax,ax
      *** 000005   e8 00 00           call __chkstk
;|***   long htoi();
;|***   printf("Please enter the hex value you want to convert:0);
; Line 6
      *** 000008   b8 00 00           mov  ax,OFFSET DGROUP:$SG12
      *** 00000b   50                 push ax
      *** 00000c   e8 00 00           call _printf
      *** 00000f   83 c4 02           add  sp,2
;|***   scanf("%s", hexvalue);
```

Note that this sample is like the object-listing sample, except that the source-program line is provided in addition to the line number.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore, as shown in the following example. This part of the listing is the same for all three kinds of listings:

```
EXTRN _printf:NEAR
EXTRN _scanf:NEAR
EXTRN __chkstk:NEAR
EXTRN __aNlmul:NEAR
EXTRN __aNNalshl:NEAR
EXTRN _hexvalue:TBYTE
```

The C Compiler automatically prefixes an underscore to all global names. If you write assembly-language routines to interface with your C program, this naming convention is important; see the section on ''Controlling the Preprocessor'' for more information.

The listing may also contain names that begin with more than one underscore (for example, *__chkstk*). Identifiers with more than one leading underscore are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *C Library Guide*. Moreover, you should avoid creating global names that begin

with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with the names reserved by the compiler.

Listing Pragmas

There are several pragmas that allow you to control the page formatting of the listings produced with the various list options. These pragmas are:

Pragma	Action
linesize	Sets the number of characters per line in the source listing.
page	Places a formfeed character(s) in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

The remainder of this section discusses each of the preceding pragmas.

The linesize Pragma

The **linesize** pragma sets the number of characters per line in the source listing. The syntax of this pragma is:

```
#pragma linesize([characters])
```

In this syntax, the optional parameter *characters* is an integer constant in the range 79-132 that specifies the number of characters you wish each line of the source listing to have. If *characters* is absent, the compiler uses the value specified in the **-Sl** option or, if that option is absent, the default value of 79 characters per line. Note that **linesize** takes effect in the line *after* the line in which the pragma itself appears.

The following example uses the pragma to produce a source listing with a 132-character line length:

```
#pragma linesize(132)
```

The page Pragma

The **page** pragma generates a formfeed (page eject) character in the source listing at the place where the pragma appears. The pragma has the following syntax:

```
#pragma page([pages])
```

The optional parameter *pages* is an integer constant in the range 1-127 that specifies the number of pages to eject. If *pages* is absent, the pragma uses a default value of 1, in which case the next line in the source file appears at the top of the next listing page.

The pagesize Pragma

The **pagesize** pragma sets the number of lines per page in the source listing. The pragma has the following syntax:

```
#pragma pagesize([lines])
```

The optional parameter *lines* is an integer constant in the range 15-255 that specifies the number of lines that you wish each page of the source listing to have. If this parameter is absent, the pragma sets the page size to the number of lines specified in the **-Sp** command-line option or, if that option is absent, to a default value of 63 lines.

The following example uses the pagesize pragma to set the number of lines per page of the source listing to 66 lines:

```
#pragma pagesize(66)
```

The skip Pragma

The **skip** pragma generates a newline (carriage return/line feed) in the source listing, at the point where the pragma appears. The pragma has the following syntax:

```
#pragma skip([lines])
```

The optional parameter *lines* is an integer constant in the range 1-127 that specifies the number of lines that you wish to skip. If this parameter is absent, **skip** defaults to one line.

The subtitle Pragma

The **subtitle** pragma sets a subtitle in the source listing. The pragma has the following syntax:

```
#pragma subtitle(subtitlename)
```

The required parameter *subtitlename* is a string literal containing the subtitle for subsequent pages in the source listing. The subtitle appears below the title on each page of the listing.

If you supply a null string ("") as the *subtitlename* parameter, **subtitle** removes any subtitle that was previously set. The **subtitlename** parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

The following statement sets the subtitle to *Error handler* for subsequent pages in the source listing:

```
#pragma subtitle("Error handler")
```

The title Pragma

The **title** pragma sets a title for the source listing. The pragma has the following syntax:

```
#pragma title(titlename)
```

The required parameter *titlename* is a string literal containing the title for the source listing. The title appears in the upper left corner of each page of the listing.

If you supply a null string ("") as the *titlename* parameter, **title** removes any title that was previously set. The **titlename** parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

The following statement sets the title to *File I/O Module* in the source listing:

```
#pragma subtitle("File I/O Module")
```

Map File

The **-Fm** option produces a map file. The map file contains a list of segments in order of their appearance within the load module. As an example, consider the following:

```
Start  Stop   Length Name                    Class
00000H 01E9FH 01EA0H _TEXT                   CODE
01EA0H 01EA0H 00000H C_ETEXT                 ENDCODE
.
.
.
```


The information in the *Start* and *Stop* columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The *Length* column gives the length of the segment in bytes, the *Name* column gives the name of the segment, and the *Class* column gives information about the segment type.

The starting address and name of each group appear after the list of segments. An example of a group listing follows:

```
Origin   Group
01EA:0   DGROUP
```

In this example, **DGROUP** is the name of the data group. **DGROUP** is the only group used for data segments by programs compiled with the C Compiler, Version 5.1.

The following map file contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name and the second is by symbol address. A maximum of 2048 symbols can be sorted in each list. (To increase the number of sorted symbols, you must specify the **-MAP** linker option with the *number* argument to create the map file; see the ''Linking with the cc Command'' chapter of this guide for details.) The notation *Abs* appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the C Compiler. These usually begin with one or more leading underscores or end with *QQ*.

```
Address          Publics by Name

003F:0096        STKHQQ
0047:1D86        _brkctl
003F:04B0        _edata
0047:0910        _end
.
.
.
0047:00EC        __abrkp
0047:009C        __abrktb
0047:00EC        __abrktbe
003F:9876  Abs   __acrtmsg
0000:9876  Abs   __acrtused
.
.
.
0047:0240        ___argc
0047:0242        ___argv

Address          Publics by Value

003F:0010        _main
003F:0047        _htoi
003F:00DA        _exp16
003F:0113        __chkstk
003F:0129        __astart
003F:01C5        __cintDIV
.
.
.
```

The addresses of the external symbols are in the "*selector*:*offset*" format, showing the location of the symbol relative to zero (the beginning of the load module).

Following the lists of symbols, the map file gives the program entry point, as shown in the following example:

```
Program entry point at 003F:0129
```

Controlling the Preprocessor

The **cc** command provides several options that control the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source-file listing.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. However, input files to the preprocessor must follow the preprocessor rules; therefore, not all arbitrary text files may be suitable for use with the preprocessor. See the *C Language Reference* for a complete discussion of C preprocessor directives and the format expected for preprocessor input.

Defining Constants and Macros (-D)

Option

-D *identifier*[=[*string*]]

The **-D** option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and *string* is its value or meaning. Note that spaces are permitted (but not required) between **-D** and the identifier.

If you leave out both the equal sign and *string*, the given constant or macro is assumed to be defined, and its value is set to 1. For example, *-DSET* is sufficient to define *SET*.

If you give the equal sign with an empty string, the given constant or macro is considered defined; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, to remove all occurrences of *register*, use the following option:

-Dregister=

Note that the identifier *register* is still considered to be defined.

The effect of using the **-D** option is the same as using a preprocessor **#define** directive at the beginning of your source file. The identifier is defined in the source file being compiled either until an **#undef** directive removes the definition or until the end of the file is reached.

You can supply a command-line definition for an identifier that is also defined within the source file. However, you must use **#undef** to remove the source-file definition, unless the source-file definition is identical to the command-line definition. The command-line definition remains in effect until the identifier is removed with an **#undef** directive.

The **-D** option is especially useful with the **#if** and **#ifdef** directives because you can control conditional-compilation directives in the source file from the command line.

Examples

cc -D NEED=2 main.c

This example defines the manifest constant *NEED* in the source file *main.c*. This definition is equivalent to placing the directive at the top of the source file as shown in the following example:

```
#define NEED 2
```

For the next example, suppose a source file named *other.c* contains the following fragment:

```
#if defined(NEED)
.
.
.
#endif
```

Suppose further that *other.c* does not explicitly define *NEED* (that is, no **#define** directive for *NEED* is present). Then all statements between the **#if** and the **#endif** directives are compiled only if you supply a definition of *NEED* by using *-D*. For instance, the following command is sufficient to compile all statements following the **#if** directive:

```
cc -DNEED main.c
```

Note that *NEED* does not have to be set to a specific value to be considered defined. The following command, in contrast, causes the statements in the **#if** block to be ignored (not compiled):

```
cc main.c
```

Predefined Identifiers (Manifest Defines)

The compiler defines several identifiers that are useful in writing portable programs. These are known as "manifest defines." You can use these identifiers to compile code sections conditionally, depending on the processor and operating system being used. They begin with "M_" for "manifest." The predefined identifiers and their functions are as follows:

Identifier	Function
M_I86	This is an Intel processor.
M_SYS3	This is Unix System III compatible.
M_SYS5	This is Unix System V compatible.
M_BITFIELDS	This compiler supports bitfields.
M_WORDSWAP	The word-within-a-longword order is swapped with respect to the DEC PDP-11.
M_UNIX	Always defined, this identifies the target operating system as an implementation of UNIX System V.
M_I*n*86	Depending on **-M0**, **-M1**, **-M2**, or **-M3**, **M_I386** is defined with 386 compiler unless **-dos** is used.
M_I86*m***M**	Always defined, this identifies the memory model, where *m* is either **S** (small model), **C** (compact model), **M** (medium model), **L** (large model), or **H** (huge model). If huge model is used, both **M_I86LM** and **M_I86HM** are defined. Small model is the default. Memory models are discussed in the ''Working with Memory Models'' chapter.
_CHAR_UNSIGNED	This is defined only when the **-J** option is given to make the **char** type unsigned by default. For more information, see the section on ''Changing the Default char Type.''
M_SDATA or **M_LDATA** **M_STEXT** or **M_LTEXT**	Depending on **-M0**, **-M1**, or **-M2**.

Removing Definitions of Predefined Identifiers (-U, -u)

Options

-U *identifier*
-u

The **-U** (for "undefine") option turns off the definition of one of the predefined identifiers discussed in the previous section. One or more spaces may separate the **-U** and *identifier*. You can specify more than one **-U** option on the same command line. The **-u** option turns off all definitions.

Example

```
cc -UM_UNIX -UM_I86 work.c
```

This example removes the definitions of two predefined identifiers. Note that the **-U** option must be given for each removal.

Producing a Preprocessed Listing (-P, -E, -EP)

Options

-P	Writes preprocessed output to a file
-E	Writes preprocessed output to standard output; includes **#line** directives
-EP	Writes preprocessed output to a file and standard output

The **-P**, **-E**, and **-EP** options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation. No object file or listing is produced, even if you specify the **-Fo** option or a listing-file option on the **cc** command line.

The **-P** option writes the preprocessed listing to a file with the same base name as the source file, but with a **.i** extension.

The **-E** option copies the preprocessed listing to the standard output (usually your terminal). It places a **#line** directive in the output at the

beginning and end of each included file and around lines removed by preprocessor commands that specify conditional compilation.

The **-E** option is useful when you want to resubmit the preprocessed listing for compilation. The **#line** directives renumber the lines of the preprocessed file, so that errors generated in later stages of processing refer to the original source file rather than to the preprocessed file.

The **-EP** option combines features of the **-E** and **-P** options; the file is preprocessed and copied both to a new file and to the standard output, but no **#line** directives are added.

Examples

```
cc -P main.c
```

This example creates the preprocessed file *main.i* from the source file *main.c*.

```
cc -E add.c > preadd.c
```

This command creates a preprocessed file with inserted **#line** directives from the source file *add.c*. The output is redirected to the file *preadd.c*.

```
cc -EP add.c
```

The command shown here produces the same preprocessed output as the second example, but without the **#line** directives. The output appears on the screen and is copied to a new file.

Preserving Comments (-C)

Option

-C

Normally, comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The **-C** (for ''comment'') option preserves comments during preprocessing. The **-C** option is valid only when the **-E**, **-P**, or **-EP** option is also used.

Example

```
cc -P -C sample.c
```

The example produces a listing named *sample.i*. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

Searching for Include Files (-I, -X)

Options

-I *directory*
-X

The **-I** and **-X** options temporarily override the default search paths for include files. (The default path is */usr/include.*)

You can add to the list of directories searched by using the **-I** (for "include") option. This option causes the compiler to search the directory or directories you specify before searching the default path */usr/include*. The space between **-I** and *directory* is optional. You can add more than one include directory by giving the **-I** option more than once in the **cc** command. The directories are searched in order of their appearance in the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs, you must restart compilation with a corrected directory specification.

You can prevent the C compiler from searching the default paths for include files by using the **-X** (for "exclude") option. When **cc** sees the **-X** option, it considers the list of standard places to be empty. This option is often used with the **-I** option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions.

Examples

```
cc -I /include -I/alt/include  main.c
```

In this example, **cc** looks for the include files requested by *main.c* in the following order: first in the directory */include*, then in the directory */alt/include*, and finally in the default directory */usr/include*.

```
cc -X -I /alt/include main.c
```

As shown in this example, the compiler looks for include files only in the directory */alt/include*. First the **-X** option tells **cc** to consider the list of standard places empty; then the **-I** option specifies one directory to be searched.

Checking for Program Errors

You may encounter several different kinds of error messages when you compile, link, and run a C program.

Several **cc** options are available to control the types of warnings generated at compile time, help with syntax checking, and verify compatibility between the actual arguments and formal parameters of a function during the early stages of program development. This section describes these options.

Understanding Error Messages

Error messages can appear at different stages of program development:

- In the compiling stage, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
- During the linking stage, the linker is responsible for generating error messages.
- During program execution, any error messages you see are run-time error messages. This category includes messages about core dumps, segmentation violations, and floating-point exceptions, which are errors generated by an 8087, 80287, or 80387 coprocessor.

Other utilities included in this package, such as the UNIX System V Link Editor (**ld**), and the **make** program-maintenance utility, generate their own error messages.

When you are compiling and linking using the **cc** command, you may see both compiler and linker messages. Compiler messages have numbers preceded by the letter *C*, and linker messages have numbers preceded by the letter *L*.

You can also distinguish the type of a message by its format. See the ''Error Messages and Exit Codes'' appendix in this guide for a description of compiler error-message formats, a list of actual compiler error messages, and explanations of the circumstances that cause them.

Compiler error messages are sent to the standard output, which is usually your terminal. If you are using the C-shell, you can redirect the messages to a file by using the standard redirection symbols at the end of your command line:

If you are using the Bourne shell, you can redirect the messages to a file by using the standard redirection syntax:

cmd > *outputfile* **2>&1**

Example

Assume the following source file is named *rm.c*:

```
#include <stdio.h>

main(argc, argv)
     int argc;
     char argv[];

     {
     register int i;
     char *name;

     for (i = 1; i < arg; ++i)
          if (unlink(name = argv[i])) {
               printf("couldn't delete %s : ", name);
               perror("");
               }
     }
```

The following C-shell command line redirects error messages to a file named *rm.err*:

```
cc rm.c >& rm.err
```

In the previous command, only output that ordinarily goes to the console screen is redirected. The error-message file *rm.err* contains the following information:

```
rm.c(11): error C2065: 'arg' : undefined
rm.c(12): warning C4047: '=' : different levels of indirection
```

Based on the errors generated, you can correct *rm.c* as shown below:

```
#include <stdio.h>

main(argc, argv)
     int argc;
     char *argv[];              /* corrects warning C4047 */

     {
     register int i;
     char *name;

     for (i = 1; i < argc; ++i)  /* corrects error C2065 */
          if (unlink(name = argv[i])) {
               printf("couldn't delete %s : ", name);
               perror("");
               }
     }
```


Setting the Warning Level (-W, -w)

Option

-W{0|1|2|3}
-w

You can suppress warning messages produced by the compiler by using the **-W** (for "warning") option. Compiler warning messages are any messages beginning with *C4*; see the "Error Messages and Exit Codes," appendix for a full listing. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend. The **-W** options affect only source files given on the command line, they do not apply to object files.

The **-W0** option turns off warning messages. This option is useful when you compile programs that deliberately include questionable statements. The **-W0** option applies to the remainder of the command line or until the next **-W** option on the command line. The **-w** option has the same effect as the **-W0** option.

The **-W1** option (the default) causes the compiler to display most warning messages.

The **-W2** option causes the compiler to display an intermediate level of warning messages. Level 2 warnings may or may not indicate serious problems. They include the following:

- Use of functions with no declared return type
- Failure to put **return** statements in functions with non-**void** return types
- Data conversions that would cause loss of data or precision

The **-W3** option displays the highest level of warning messages, including warnings about the uses of non-ANSI features and extended keywords and about function calls before the appearance of function prototypes in the program.

Note that the warning messages in the "Error Messages and Exit Codes" appendix indicate the warning level that must be set (that is, the number for the appropriate **-W** option) for the message to appear.

Example

```
cc -W3 crunch.c print.c
```

This example enables all possible warning messages when the *crunch.c* and *print.c* source files are compiled.

Checking Syntax (-Zs)

Option

-Zs

The **-Zs** option causes the compiler to perform only a syntax check on the source files that follow the option on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file.

When you give the **-Zs** option, the compiler does not generate code or produce object files, object listings, or executable files. However, the compiler does display error messages if the source file has syntax errors. You can specify the **-Fs** option on the same command line to generate a source listing that shows these error messages. For more information about the **-Fs** option, see the section on "Types of Listings."

Example

```
cc -Zs test*.c
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with *test* and end with the default extension (**.c**). The compiler displays messages for any errors found.

Generating Function Declarations (-Zg)

Option

-Zg

The **-Zg** option generates a function declaration for each function defined in the source file. You can use the -Zg option with multiple source files. The function declaration includes the function return type and an argument type list created from the types of the formal parameters of the function. Any function declarations already encountered are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using shell redirection.

When the **-Zg** option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type-checking. The presence of a declared argument-type list for a function ''turns on'' the compiler's type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type-checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type-checking is a recent addition to the C language, so many existing C programs will not have argument-type lists. See the *C Language Reference* for more information about function declarations and argument-type lists.

You can use the **-Zg** option even if your source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not conflict. No conflict occurs when one declaration has an argument-type list and another declaration of the same function does not, as long as the return types are identical.

Note

If you use the **-Zg** option and your program contains formal parameters that have structure, enumeration, or union type (or pointers to such types), then the declaration for each structure, enumeration, or union type must have a tag. For example, use the following form:

```
struct tagA {
            .
            .
            .
      } A;
```

Example

```
cc -Zg file.c > filedecls.h
```

This command causes the compiler to generate argument-type lists for functions defined in *file.c*. The list of declarations is redirected to *filedecls.h*.

Preparing for Debugging (-Zi, -Od)

Options

-Zi Creates object file for use with the source-level debugger, **sdb and CodeView**
-Od Disables code optimization to help with debugging

The **-Zi** option produces an object file containing full symbolic-debugging information for use with the source-level debugger. This object file includes full symbol-table information and line numbers. If the **-Zi** option is given with no explicit **-O** options, all optimizations involving code motion and rearrangement are suppressed, although simple optimizations are still performed. If any explicit **-O** options are given, *all* requested optimizations are performed.

The **-Od** option tells the compiler not to perform most optimizations. Some peephole optimizations and other simple optimizations are still performed. (Without the **-Od** option, the default is to optimize.) You may want to use this option when you plan to use a symbolic debugger with your object file, since optimization can involve rearrangement of instructions that make it difficult for you to recognize and correct your code when debugging. However, turning off optimizations may increase the size of the code generated to the point where it might not be possible to link your program.

Other optimization options are discussed in the section on ''Optimizing.''

Example

```
cc -Zi -Od test.c
```

This command produces an object file named *test.o* that contains line numbers corresponding to the line numbers of *test.c*. A source-listing file *test.lst* is also created. Limited optimization is performed.

Optimizing

The optimizing capabilities available with the C Compiler can reduce the storage space or execution time required for a program. This is achieved by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the **-O** options, the **loop_opt** pragma (described in the section on ''Loop Optimization''), the **intrinsic** pragma, and the **function** pragma (described in the section ''Generating Intrinsic Functions'') to exercise greater control over the optimizations performed. In addition, you can use the **-Gs** option or **check_stack** pragma to reduce program size and speed up execution.

Controlling Optimization (-O Options)

Option

-O*string*
#pragma loop_opt([on|off])
#pragma intrinsic(*function1*[**,***function2*]**...)**
#pragma function(*function1*[**,***function2*]**...)**

The **-O** options give you control over the optimization procedures that the compiler performs. One or more of the letters in *string* following the **-O** let you choose how the compiler performs optimization:

Letter	Optimizing Procedure
none	Performs optimization equivalent to **-Oct**
a	Relaxes alias-checking
c	Eliminates common expressions
d	Disables optimization
i	Expands certain intrinsic functions inline
l	Enables loop optimization
p	Improves consistency of floating-point results
s	Favors code size during optimization
t	Favors execution speed during optimization
x	Maximizes optimization (equivalent to **-Oatcli**)

The letters can appear in any order; for example, **-Oat** and **-Ota** have the same effect. More than one **-O** option can be given; the compiler uses the last **-O** option given if any conflict arises. Each option applies to all source files following that option on the command line.

The following sections discuss the various optimization options and their effects.

Relaxing Alias Checking (-Oa)

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of ''aliases'' in the program. Aliases are multiple names (that is, symbolic references) for the same memory location in a program. Most commonly, aliases occur as a result of code similar to that shown in the following example:

```
func()
{
int  x, *p;

p = &x;    /* now "x" and "*p" refer to the same */
          /* memory location                    */
.
.
.
}
```


Use of the **-Oa** option can reduce the size of executable files and speed program execution. Its use is especially recommended when you also specify the **-Ol** option, since the compiler can detect a number of loop optimizations when the **-Oa** option is in effect that it cannot detect when **-Oa** is not in effect. However, before you specify **-Oa**, you must make sure that your program does not use aliases either directly or indirectly.

Note

Exercise caution when using the **-Od** option, because responsibility for alias checking is trasferred to the programmer from the compiler.

The use of aliases is important only if both names are actually used to reference the memory location. The following example illustrates the use of aliases:

```
func()
{
        int  x, *p;

        p = &x;

        .
        .
        .
/* ...expressions involving only *p */
        .
        .
        .
}
```

Since all access to the memory location labeled *x* is through the pointer *p*, *x* has no significance in the function. To illustrate, *func* could be rewritten as the following pair of functions:

```
func1()
{
        int  x;

        func2(&x);
}

func2(p)

int  *p;
{
        .
        .
        .
/*  ...expressions involving *p  */
          .
          .
          .
}
```

In this equivalent form, the alias created in *func1* is insignificant, since the memory location is not referenced at all and *func2* does not use aliases since *x* is not even in the scope of the function. The **-Oa** option can be safely specified in compiling either of these equivalent forms.

In addition to the obvious cases discussed above, aliases can be created through the use of pointers in other, more subtle ways. Two such cases involving the use of pointers as function arguments are illustrated in the following example:

```
int  x;

func(p)

int  *p;
{
        .
        .
        .
/*  ...expressions involving *p and x  */
        .
        .
        .
}
```


In this example, *x* is a communal variable, so the function can be called with *func(&x)*. The **-Oa** option can be used safely only if it is known that *func* is never invoked with the address of *x* as an argument.

```
func(p1, p2)

int *p1, *p2;
{
        .
        .
        .
/* ...expressions involving *p1 and *p2 */
        .
        .
        .
}
```

In this example, the function may be invoked with the same value for both arguments (that is, *func(p,p)* or *func(&x,&x)*). Thus, the **-Oa** option can be safely specified only if it is known that the function is always called with distinct values for the two arguments.

One use of aliases occurs so frequently that a special provision has been made for it. When the compiler encounters a call to a function with address-type arguments, it always assumes that all variables whose addresses are passed to the function are modified. If such function calls appear in a program, the **-Oa** option can be specified safely even though the function call results in an alias for each variable whose address is passed. The following example illustrates how the compiler handles this case:

```
func1()
{
        int  x, y, a, b;
        .
        .
        .
        x = a + b;

func2(&a);

        y = a + b;
}
```


As shown, when the compiler encounters the function call *func2(&a)*, it assumes that the function modifies *a*, even if the **-Oa** option has been specified. The compiler generates code to evaluate each instance of the expression *a + b*, rather than eliminating a common subexpression incorrectly.

Although you should convert programs that use aliases if you plan to compile them with the **-Oa** option, it is helpful to know the units of a program where the optimizations affected by the use of **-Oa** are applied. This information indicates where the uses of aliases are most likely to cause incorrect optimizations if **-Oa** is specified. The following list describes the program units where such optimizations are performed:

- All of the C optimizations, except for loop optimizations, that may be affected by the incorrect use of **-Oa** are applied at the level of basic blocks. In the C Compiler, the **-Oa** option can generally be used even if aliases are employed, provided no memory location is referenced by more than one name within any basic block. (A "basic block" is a contiguous sequence of statements, with a unique entry point and exit point and no branching in between. In C programs, basic blocks most often appear as the clauses of **if** statements, **switch** statements, loop bodies, or function bodies, although they may also occur as sequences of statements delimited by user labels.)

- Loop optimizations are applied at the level of whole loop bodies. Thus, if loop optimization is enabled, **-Oa** can generally be used even if aliases are employed, provided that no memory location is referenced by more than one name within any basic block or loop body.

Disabling Optimization (-Od)

The **-Od** option turns off most optimizations. This is useful in the early stages of program development to avoid optimizing code that will later be changed. Because optimization may involve rearrangement of instructions, you may also want to specify the **-Od** option when you use a debugger with your program or when you want to examine an object-file listing. If you optimize before debugging, it can be difficult to recognize and correct your code. However, note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Generating Intrinsic Functions (-Oi)

The **-Oi** option tells the compiler to generate intrinsic functions instead of function calls for certain functions. Intrinsic functions may be in-line functions, may use special argument-passing conventions, or (in some cases) may do nothing. Programs that use intrinsic functions are faster because they do not include the overhead associated with function calls. However, they may be larger because of the additional code that is generated.

The following functions have intrinsic forms:

- **memset**, **memcpy**, and **memcmp**
- **strset**, **strcpy**, **strcmp**, and **strcat**
- **outp**
- **_rotl**, **_rotr**, **_lrotl**, and **_lrotr**
- **min**, **max**, and **abs**

Note

Intrinsic versions of the **memset**, **memcpy**, and **memcmp** functions in compact- and large-model programs cannot handle huge arrays or huge pointers. To use huge arrays or huge pointers with these functions, you must compile your program with the huge memory model (that is, using the **-Mh** option on the command line).

You can use the **intrinsic** pragma to generate intrinsic functions only for selected functions. This pragma has the following format:

#pragma intrinsic (*function1* [,*function2*]...)

The **intrinsic** pragma affects the specified functions from the point where the pragma appears until either the end of the source file or the next **function** pragma specifying any of the same functions. The **function** pragma has the following format:

#pragma function (*function1* [,*function2*]...)

Note that you can also use the **function** pragma selectively to generate function calls instead of intrinsic functions when you compile a program with the **-Oi** option.

Loop Optimization (-Ol)

The **-Ol** option tells the compiler to perform loop optimizations. For best performance, the **-Ol** option should be specified along with the **a** option letter (**-Oal**), since the compiler can detect more loop optimizations when it relaxes its assumptions about the use of aliases.

You can use the **loop_opt** pragma to turn loop optimization on or off for selected functions. When you want to turn off loop optimization, put the following line before the code on which you don't want to perform loop optimization:

```
#pragma loop_opt (off)
```

Note that the preceding line disables loop optimization for all code that follows it in the source file, not just the routines on the same line. To reinstate loop optimization, insert the following line:

```
#pragma loop_opt (on)
```

If no argument is given to the **loop_opt** pragma, loop optimization reverts to the behavior specified on the command line: enabled if the **-Ox** or **-Ol** option is in effect, and disabled otherwise. The interaction of the **loop_opt** pragma with the **-Ol** and **-Ox** options is explained in greater detail in Table 2.5.

Table 2.5
Using the loop_opt Pragma

Syntax	Compiled with -Ox or -Ol?	Action
#pragma loop_opt()	no	Turns off optimization for loops that follow
#pragma loop_opt()	yes	Turns on optimization for loops that follow
#pragma loop_opt (on)	yes or no	Turns on optimization for loops that follow
#pragma loop_opt (off)	yes or no	Turns off optimization for loops that follow

Achieving Consistent Floating-Point Results (-Op)

The **-Op** option is useful when floating-point results must be consistent within a program. This option changes the way in which the program handles floating-point values.

Ordinarily the compiler stores each floating-point value in an 80-bit register. In subsequent references to that value, the compiler reads the value from the register. When the final value is written to memory, it is truncated, since floating-point types are allocated fewer than 80 bits of storage (32 bits for the **float** type and 64 bits for the **double** type). Thus, the value stored in the register may actually be more precise than the same value stored in a floating-point variable. Since the value is truncated each time it is written to memory, over the course of the program the value stored in the machine register may become quite different from the value that is written to memory.

If you use the **-Op** option, when floating-point values are referenced, the compiler reloads them from floating-point variables rather than from registers. Using **-Op** gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more control over the truncation (and hence the consistency) of floating-point values.

Optimizing for Speed and Code Size (-Ot, -Os)

When you do not give a **-O** option to the **cc** command, it automatically uses **-Ot**, meaning that program-execution speed is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when the **-Ot** option is in effect, the compiler generates intrinsic functions to perform shift operations on long operands.

To cause the compiler to favor smaller code size instead, use the **-Os** option. For example, when the **-Os** option is in effect, the compiler uses function calls to perform shift operations on long operands.

Producing Maximum Optimization (-Ox)

The **-Ox** option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
-Oatcli
```

That is, the **-Ox** option relaxes alias checking, generates all intrinsics for the functions listed in the section "Generating Intrinsic Functions," performs loop optimizations, favors execution time over code size, and removes stack probes. Note that the interactions between the **-Ox** option and the **loop_opt** pragma are the same as those described in Table 2.5. For more information about stack probes and ways of controlling their use, see the following section, "Removing Stack Probes."

Examples

```
cc -Oal file.c
```

This command tells the compiler to perform loop optimizations and relax alias-checking when it compiles *file.c*. The compiler favors program speed over program size, since the **-Ot** option is also specified by default.

```
cc -c -Os file.c
```

This command favors code size over execution speed when *file.c* is compiled.

```
cc -Od *.c
```

This command compiles and links all C source files with the default extension (**.c**) in the current directory and disables optimization. This command is most useful during the early stages of program development, since it improves compilation speed.

Removing Stack Probes (-Gs)

Options

-Gs
#pragma check_stack([on|off])

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this either with the **-Gs** option or with the **check_stack** pragma.

A ''stack probe'' is a short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function. The stack probe routine is called at every function entry point. Ordinarily, the stack probe routine generates a stack overflow message when it determines that the required stack space is not available. When stack-checking is turned off, the stack probe routine is not called, and stack overflow can occur without being diagnosed (that is, no error message is printed).

Use the **-Gs** option when you want to turn off stack-checking for an entire module if you know that the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls, or that have only modest local variable requirements. In the absence of the **-Gs** option, stack-checking is on.

Use the **check_stack** pragma when you want to turn stack-checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the **-Gs** option) for the rest. When you want to turn off stack-checking, put the following line before the definition of the function you don't want to check:

```
#pragma check_stack (off)
```

Note that the preceding line disables stack-checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack-checking, insert the following line:

```
#pragma check_stack (on)
```

Note

For earlier versions of C, the **check_stack** pragma had a different format: **check_stack+** to enable stack-checking and **check_stack-** to disable stack-checking. Although the C Compiler still accepts this format, its use is discouraged, since it may not be supported in future versions.

If no argument is given for the **check_stack** pragma, stack-checking reverts to the behavior specified on the command line: disabled if the **-Gs** option is given, or enabled otherwise. The interaction of the **check_stack** pragma with the **-Gs** option is explained in greater detail in Table 2.6.

Table 2.6

Using the check_stack Pragma

Syntax	Compiled with -Gs Option?	Action
#pragma check_stack()	yes	Turns off stack-checking for routines that follow
#pragma check_stack()	no	Turns on stack-checking for routines that follow
#pragma check_stack(on)	yes or no	Turns on stack-checking for routines that follow
#pragma check_stack(off)	yes or no	Turns off stack-checking for routines that follow

Note

The **-Gs** option should be used with care. Although it can make programs smaller and faster, it may mean that the program will not be able to detect certain execution errors.

Example

```
cc -Oals -Gs file.c
```

This example optimizes the file *file.c* by removing stack probes with the **-Gs** option. The letters specified with the **-O** option tell the compiler to relax alias-checking (**a**), perform loop optimization (**l**), and favor code size over program speed (**s**). If you want stack-checking for only a few functions in *file.c*, you can use the **check_stack** pragma around the definitions of functions you want to check. Similarly, if you want to perform loop optimization on only a few functions in *file.c*, you can use the **loop_opt** pragma around the definitions of functions on which you want to perform loop optimization.

Enabling/Disabling Language Extensions (-Ze, -Za)

Option

-Ze	Enables language extensions (default)
-Za	Disables language extensions (strict ansi specifications)

The C Compiler is moving to support the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These additional features are enabled when the **-Ze** (default) option is in effect and disabled when the **-Za** option is in effect. They include the following:

- The **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** keywords
- Use of casts to produce values, as in this example:

```
int *p;
((long *)p)++;
```

The preceding example could be rewritten to conform with ANSI C as shown here:

```
p = (int *)((char *)p + sizeof(long));
```

- Redefinitions of **extern** items as **static**, as follows:

```
extern int foo();
static int foo()
{}
```

- Use of trailing commas (,) without ellipses (,...) in function declarations to indicate variable-length argument lists, such as:

```
int printf(char *,);
```

- Benign **typedef** redefinitions within the same scope, like this:

```
typedef int INT;
typedef int INT;
```

- Use of mixed character and string constants in an initializer, for instance:

```
char arr[5] = {'a', 'b', "cde"};
```

- Use of bit fields with base types other than **unsigned int** or **signed int**

Use the **-Za** option if you will be porting your program to other environments. The **-Za** option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed previously.

Packing Structure Members (-Zp)

Option

-Zp[{1|2|4}]
#pragma pack([{1|2|4}])

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte-aligned.

- Structures are word-aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word-aligned.

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The **-Zp** option and the **pack** pragma control how structure data are ''packed'' into memory.

Use the **-Zp** option when you want to specify the same packing for all structures in a module. When you give the **-Zp[*n*]** option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries, depending on the option you choose. If you use the **-Zp** option without an argument, structure members are packed on 1-byte boundaries.

On some processors, the **-Zp** option may result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with **int** or **long** type are packed in such a way that they begin on odd-byte boundaries.

Use the **pack** pragma when you want to specify packing other than that specified on the command line for particular structures. Give the **pack(*n*)** pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the **pack()** pragma with no arguments.

Table 2.7 shows the interaction of the **-Zp** option with the **pack** pragma.

Table 2.7

Using the pack Pragma

Syntax	Compiled with -Zp Option?	Action
#pragma pack()	yes	Reverts to packing specified on the command line for structures that follow
#pragma pack()	no	Reverts to default packing for structures that follow

#pragma pack(*n*)	yes or no	Packs the following structures to the given byte boundary until changed or disabled

Example

```
cc -Zp prog.c
```

This command causes all structures in the program *prog.c* to be stored without extra space for alignment of members on **int** boundaries.

Setting the Stack Size (-F)

Option

-F *hexnum*

The **-F** option sets the size of the program stack. A space must separate the **-F** and *hexnum*. (This option applies only to the 286 compiler.)

The *hexnum* is a hexadecimal value representing the stack size in bytes. The value must be less than 0xFFFF hexadecimal (65,535 decimal).

If you do not specify this option, the start-up routine in the standard C library sets the default stack size to 2K.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

The **-F** option is a linking option that affects executable files only; it does not have any effect on source or object files.

Example

```
cc -F C00 *.o
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all of the object files in the current working directory.

Restricting the Length of External Names (-nl)

Option

-nl *number*

The **cc** command allows you to restrict the length of external (public) names by using the **-nl** option. The *number* is an integer specifying the maximum number of significant characters in external names. The space between **-nl** and *number* is optional.

When you use the **-nl** option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters, but the extra characters are simply ignored.

The **-nl** option is typically used to conserve space or to aid in creating portable programs. The C Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

Labeling the Object File (-V)

Option

-V *string*

Use the **-V** (for ''version'') option to embed a text string in an object file. The *string* must be enclosed in double quotation marks (" ") if it contains white-space characters or embedded double quotation marks. A backslash (\) must precede any embedded double quotation marks.

A typical use of the **-V** option is to label an object file with a version number or copyright notice.

Example

```
cc -V "C Compiler Version 5.1" main.c
```

This command places the following string in the object file *main.o*:

```
C Compiler Version 5.1
```

Changing the Default char Type (-J)

Option

-J

In C, the **char** type is signed by default, so if a **char** value is widened to **int** type, the result is sign-extended. You can change this default to **unsigned** with the **-J** option, causing the **char** type to be zero-extended when widened to **int** type. However, if a **char** value is explicitly declared **signed**, the **-J** option does not affect it, and the value is sign-extended when widened to **int** type.

When you specify **-J**, the compiler automatically defines the identifier **_CHAR_UNSIGNED**.

Controlling the Calling Convention (-Gc)

Options

-Gc
fortran
pascal
cdecl

The **-Gc** option and the **fortran**, **pascal**, and **cdecl** keywords allow you to control the function-calling and naming conventions so that your C programs can call and be called by functions that are written in FORTRAN and Pascal.

Because C, unlike other languages such as Pascal and FORTRAN, allows you to write functions that take variable numbers of arguments, it must handle function calls differently. Languages such as Pascal and FORTRAN normally push actual parameters to a function in left-to-right order, with the last argument in the list being the last one pushed on the stack. In contrast, C functions do not always know the number of actual parameters, so they must push their arguments from right to left, with the first argument in the list being the last one pushed.

Additionally, the calling function must remove the arguments from the stack in C (rather than having the called function do it, as in Pascal and FORTRAN). If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling

function (as in C), it appears every time there is a function call. Since function calls are more numerous than individual functions, the Pascal/FORTRAN method is slightly smaller and more efficient.

The C Compiler has the ability to generate the Pascal/FORTRAN calling convention in one of several ways. The first is through the use of the **pascal** and **fortran** keywords. When these keywords are applied to functions, or to pointers to functions, they indicate a corresponding Pascal or FORTRAN function. Therefore, the correct calling convention must be used. In the following example, *sort* is declared as a function using the alternative calling convention:


```
short pascal sort(char *, char *);
```

The **pascal** and **fortran** keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

The second method for generating the Pascal/FORTRAN calling convention is to use the **-Gc** option. If you do this, the entire module is compiled using the alternative calling convention. You might use this method to make it possible to call all the functions in a C module from another language, or to gain the performance and size improvement provided by this calling convention. When you use **-Gc** to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Thus, using **-Gc** would normally mean that you cannot call or define functions that take variable numbers of parameters, and that you cannot call functions such as the C library functions that use the C calling sequence.

To overcome these restrictions, the **cdecl** keyword has been added to C. This keyword is the ''inverse'' of the **fortran** and **pascal** keywords. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal C calling convention. This allows you to write C programs which take advantage of the more efficient calling convention while still having access to the entire C library, other C objects, and even user-defined functions that can take variable-length argument lists.

If you compile with the **-Gc** option, either you must declare the **main** function in the source program with the **cdecl** keyword, or you must change the start-up routine so that it uses the correct naming and calling conventions when calling **main**.

Run-time library functions all use the C calling convention. Therefore, care must be taken to declare them **cdecl** functions.

Use of the **pascal** and **fortran** keywords, and the **-Gc** option also affects the naming convention for the associated item or items; the name is converted to uppercase (capital letters), and the leading underscore that C normally prefixes is not added. The **pascal** and **fortran** keywords can be applied to data items and pointers, as well as functions. When applied to data items or pointers, these keywords force the naming convention described above for that item or pointer.

The **pascal**, **fortran**, and **cdecl** keywords, like the **near**, **far**, and **huge** keywords, are disabled by use of the **-Za** option. If this option is given, these names are treated as ordinary identifiers rather than keywords.

Examples

```
int cdecl var_print(char*,...);
```

In this example, *var_print* is allowed to have a variable number of arguments by declaring it as a function using the normal right-to-left C function calling convention and naming conventions. The *cdecl* keyword overrides the left-to-right calling sequence set by use of the **-Gc** option when compiling the source file in which this declaration appears. If this file is compiled without the **-Gc** option, *cdecl* has no effect since it is the same as the default C convention.

```
float *pascal nroot(number, root)
```

This instruction declares *nroot* to be a function returning a pointer to a value of type *float*. The function *nroot* uses the default calling sequence (left-to-right) and naming conventions for FORTRAN and Pascal programs.

```
long pascal index
```

This example simply changes the naming convention for the data item *index*: it is included in the object file in all capital letters and without a leading underscore.

Compiling Programs for DOS Environment (-dos, -FP)

The C compiler is capable of compiling programs that will execute in the DOS environment.

The **-dos** option instructs the compiler to use the set of libraries in */usr/lib/dos* and to use a different linker. Note that programs compiled with **-dos** will not run in the UNIX System V environment. Also note that many UNIX System V system calls are not supported in DOS.

There are a variety of **-FP** options that can be used along with **-dos** to control floating-point operations. For more information on **-FP** and on DOS cross-development in general, see "The DOS-OS/2 Development Guide" in the *Developer's Guide*, and the chapter "C Language Portability" in this guide.

Displaying Compiler Passes (-d, -z)

The **cc** command is actually a driver program which executes a series of compiler passes, perhaps an assembler pass, and a linker. It collects the various options and files on its command line and distributes them to the proper pass or to the linker. The C compiler is conceptually a four-pass compiler. The function of the various compiler passes is outlined below:

Pass 0
Pass zero of the compiler is comprised of the preprocessor and parser. The preprocessor handles file inclusion, macro expansion, and text substitution, and allows you to define constructs for conditional compilation. The parser performs two functions: (1) building a context-free grammar tree to pass to Pass 1; and (2) constructing a symbol table.

Pass 1
Pass two generates code. It walks the grammar tree constructed by pass 0, applies semantic rules to each syntactic construct, and produces the binary code indicated by the semantic rules.

Pass 2
The third pass provides post-generation optimization. It analyzes the code generated by pass 1 and applies optimization rules to alter the code for better performance (such as elimination of redundant code, rearrangement, etc.). It creates the object code and outputs listing files (if requested).

The **-d** option displays the various passes and their arguments before they are executed. The **-z** option shows the passes but does not execute them.

Producing OMF Object and Executable Files (-xenix)

By default, **cc** produces object and executable files using the COFF format, which is the same format used by the AT&T development system. The **-xenix** option causes **cc** to produce object and executable files that use the OMF format, which is compatible with the XENIX System V development system tools. When the **-xenix** option is used with any of the options that produce assembly-language output, the warning message normally issued (**masm** directives) is suppressed. Note that UNIX System V can execute programs that use *either* COFF or OMF formats.

Miscellaneous Pragmas

The following pragmas allow you to embed comments in the object or executable file or to send a string to the standard output:

Pragma	Action
comment	Places a comment record in the object file.
Message	Sends a message string to the standard output.

The comment pragma

The **comment** pragma allows you to place a comment record in an object file or executable file. The pragma has the following syntax:

```
#pragma comment (commenttype [, commentstring])
```

The required parameter *commenttype* specifies the type of comment record. The optional *commentstring* parameter is a string literal that provides additional information for some comment types. The following table lists and describes the types of comment records accepted by the **comment** pragma.

Record	Description
compiler	Places the name and version number of the compiler into the object file. This comment record is ignored by the linker. If you supply a *commentstring* parameter for this record type, the compiler generates a warning message.
exestr	Places the string specified in *commentstring* into the object file. At link time, this string is placed into the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.
lib	Places a library-search record into the object file. This comment type must be accompanied by a *commentstring* containing the name (possibly including the path) of the library that you want the linker to search for. Since the library name precedes the default library-search records in the object file, the linker searches for this library just as if you had named it on the command line. You can place multiple library-search records in the same source file. Each record appears in the object file in the same order it is encountered in the source file.
user	Places a general comment into the object file. The *commentstring* parameter contains the text of the comment. This comment record is ignored by the linker.

The following examples illustrate some of the uses of the **comment** pragma. The following pragma causes the linker to search for the library **mylibry.a**. The linker searches first in the current working directory and then in the path specified in the **LIB** environment variable:

```
#pragma comment(lib,mylibry)
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

```
#pragma comment(compiler)
```

For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:


```
#pragma comment(user, "Compiled on " __DATE__ "at " __TIME__)
```

The message Pragma

The **message** pragma sends a string to the standard output. The pragma has the following syntax:

```
#pragma message(messagestring)
```

The *messagestring* parameter is a string literal that contains the message that you wish to send to the standard output. This pragma does not cause termination of the compilation. A typical use of **message** is to display informational messages at compile time.

The following code fragment uses **message** to display a message during compilation:

```
#if M_I86MM
     #pragma message("Medium memory model")
#endif
```

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. For example, the following statement displays the name of the file being compiled and the date and time when the file was last modified:

```
#pragma message( "Compiling " __FILE__ ". Last modified: " __TIMESTAMP__)
```

Predefined Macro Names

The C Compiler supports all of the predefined macro names found in the ANSI proposed standard for the C language. These provide a convenient means for obtaining the date and time of the compilation and for indicating whether the compiler purports to conform fully to the proposed ANSI standard. The __TIMESTAMP__ identifier offers a capability not found in the proposed ANSI standard. The following list explains each of these names:

Macro Name	Description
__DATE__	The date of compilation, expressed as a string literal in the form: Mmm [d]d yyyy.
__FILE__	The file name, expressed as a string literal.
__LINE__	The program line number where the __LINE__ macro was used.
__STDC__	The integer constant 0. If equal to 1, this macro indicates full conformity with the ANSI proposed standard for the C programming language.
__TIME__	The time of compilation, expressed as a string literal in the form: hh:mm:ss.
__TIMESTAMP__	The date and time of the last modification of the source file, expressed as a string literal in the form: Ddd Mmm [d]d hh:mm:ss yyyy

The __TIMESTAMP__ macro name is not ANSI standard. Note that its time and date indicate the last modification of the source file, whereas __DATE__ and __TIME__ indicate the time of compilation. You can find additional information about the ANSI-compatible predefined identifiers __DATE__ and __TIME__ in the *C Language Reference*.

The following code fragment uses three predefined macros with the **#message pragma** to display informational messages at the time of compilation.

```
#pragma message("Compilation date: "__DATE__)
#pragma message("Compiling: "__FILE__)
#pragma message("Last modification: "__TIMESTAMP__)
```


Here is the output you might see from the preceding code fragment:

```
Compilation date: Dec  2 1987
Compiling: sample.c.
Last modification: Mon Dec  1 12:02:51 1987
```

Chapter 3

Linking with the cc Command

Introduction

Since the **cc** command controls linking as well as compiling, you can specify linker options and libraries other than the default combined library to be linked with your object files on the **cc** command line.

The Default Linking Process

When the **cc** command compiles a source file, it encodes the name of the appropriate library in the object file. The library name embedded in the library file is determined by the memory-model (**-M**) option you give on the **cc** command line.

If you use the default memory-model option (**-Ms**), **cc** encodes the name of the standard library that corresponds to the defaults.

When an object file is linked, the linker looks for libraries matching the names encoded in the object file.

The result is that you do not ordinarily need to give library names on the **cc** command line. For descriptions of the situations that require you to specify libraries on the **cc** command line, see the "Specifying Libraries" section in this chapter.

The linker used is /bin/ld, which by default produces object files in the COFF format. If the object file uses the OMF format, you have two choices:

1. Use **cvtomf** to convert the object file to COFF format. This occurs automatically if you do not use the **-xenix** option.

2. Direct the linker, **/bin/ld**, to link the file and produce an executable file using the *x.out* format. This occurs automatically if you use the **-xenix** option.

The remainder of this section applies only to the XENIX System V linker, **ld(CP)**. The AT&T linker is also described on the manual page, **ld(CP)**.

Passing Linker Information: The -link Option

To pass linker options or nondefault library names to the linker, give the following options on the **cc** command line after any source- and object-file names and **cc** options:

-link

Use the rest of the command line to specify linker options, libraries, and library search paths. Note that library names can also be specified with source- and object-file names before the **-link** option on the command line, as long as the library names have the **.a** extension. These library names are searched before library names specified after the **-link** option. Refer to the following sections for more information:

- "Specifying Libraries," to learn about specifying libraries and library search paths
- "Specifying Linker Options," for descriptions of the linker options that apply to C.

If you use the **-link** option with the **cc** command, it must be the last option on the command line.

Specifying Libraries

To link object files with libraries other than the default library, give the names of the nondefault libraries on the **cc** command line. Library names appearing before **-link** must have the **.a** extension; library names appearing after **-link** may have blank extensions or no extensions.

Since the object file already contains the names of the correct combined library, you do not need to specify libraries unless you want to do any of the following:

- Link with additional libraries
- Look for libraries in different locations
- Override the use of the default library

Linking with Additional Libraries

If you specify additional libraries to **cc**, the linker searches the libraries you specify *before* it searches the default library to resolve external references in the object files. It searches the libraries you specify in their order of appearance on the command line.

If a library name includes a path specification, the linker searches only that path for the library.

If you specify only a library name (without a path specification), the linker searches in the following locations to find the given library file:

- The current working directory
- Any path specifications that you give, in their order of appearance on the command line
- The default location */lib* or */lib/386*

If a library name without an extension appears after the **-link** option, the linker automatically supplies the **.a** extension. If you want to link a library file with an extension other than **.a**, you must specify the complete library name.

Looking in Different Locations for Libraries

You can tell the linker to look in different locations for libraries by giving a path specification on the **cc** command line.

The linker looks for the default libraries in the same order as it looks for libraries given on the command line.

Specifying Linker Options

When you use the **cc** command to invoke the linker, any linker options you specify (other than those supported by **cc** options such as **-F** and **-Fm**) must appear after the **-link** option on the command line. All options begin with the dash (**-**).

The following sections outline the rules for specifying linker options on the **cc** command line.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique, so that the linker can determine which option you want. The minimum legal abbreviation for each option is indicated in the syntax of the option.

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

Some linker options take numerical arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65,535
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with 0. For example, the number *10* is a decimal number, but the number *010* is an octal number, equivalent to 8 in decimal
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with 0x or 0X. For example, *0x10* is a hexadecimal number, equivalent to 16 in decimal

Linker Options

This section summarizes some of the linker options that can be used with C programs. Note that this section does not describe all available linker options. For a complete list, refer to the **ld**(CP) manual page in the *Programmer's Reference*.

The following linker option is most commonly used with C programs:

-SE[GMENTS]:*number*

Controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1-1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. When you set the segment limit higher than 128, the linker allocates more space for segment information. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting *number* to reflect the actual number of segments in the program. The linker displays an error message if the number of segments allocated is too high for the amount of memory the linker has available.

The following linker options can be used with C programs, but they perform the same actions as **cc** options. Therefore, you do not need to use them unless you are compiling and linking in separate steps.

-M[AP][:*number***]**

Creates a map file. This option is equivalent to using the **-Fm** option with the **cc** command, except that you can give a *number* argument with the **-M** option. The *number* argument is any positive integer (decimal, octal, or hexadecimal) up to 65,535 (decimal) specifying how many symbols are sorted in the map listing. If no *number* argument is given, a maximum of 2048 symbols is sorted. (In practice, the number of sorted symbols is limited by the amount of free heap space.) If a *number* argument is given, the alphabetical list of symbols does not appear in the map file.

-LI[NENUMBERS]

Creates a map file and includes the line numbers and associated addresses of the source program. This option is equivalent to using the **-Zd** option with the **cc** command. For more information about the **-Zd** option, see the ''Compiling with the cc Command'' chapter of this guide.

-ST[ACK]:*number*

Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. This option is equivalent to using the **-F** option of the **cc** command. For more information about the **-F** option, see the ''Compiling with the cc Command'' chapter of this guide.

Chapter 4

Running C Programs on System V